TITLE:

A Set Covering Approach for the Pickup and Delivery Problem with Additional Constraints (Numerical Optimization methods, theory and applications)

AUTHOR(S):

Hashimoto, Hideki; Ezaki, Youichi; Ygiura, Mutsunori; Nonobe, Koji; Ibaraki, Toshihide; Lokketangen, Arne

CITATION:

Hashimoto, Hideki ...[et al]. A Set Covering Approach for the Pickup and Delivery Problem with Additional Constraints (Numerical Optimization methods, theory and applications). 数理解析研究所講究録 2008, 1584: 162-174

ISSUE DATE:

2008-02

URL:

http://hdl.handle.net/2433/81490

RIGHT:

京都大学学術情報リポジトリ
KURENAI 紅
Kyoto University Research Information Repository

数理解析研究所講究録
第1584巻 2008年 162-174

A Set Covering Approach for the Pickup and Delivery Problem with Additional Constraints

京都大学大学院情報学研究科　橋本 英樹 (Hideki Hashimoto)
Graduate School of Informatics, Kyoto University
キヤノンシステムソリューションズ　江崎 洋一 (Youichi Ezaki)
Canon System Solutions Inc.
名古屋大学大学院情報科学研究科 柳浦 睦憲 (Mutsunori Yagiura)
Graduate School of Information Science, Nagoya University
法政大学デザイン工学部　野々部 宏司 (Koji Nonobe)
Graduate School of Art and Technology, Hosei University
関西学院大学理工学部　茨木 俊秀 (Toshihide Ibaraki)
School of Science and Technology, Kwansei Gakuin University
Molde College　Arne Løkketangen

Abstract

In this paper, we generalize the pickup and delivery problem with time windows by allowing additional constraints on each route, and propose a heuristic algorithm. Our algorithm first generates a set of feasible routes, and repeats modifying the set by using the information from a Lagrangian relaxation of the set covering problem corresponding to the current set. It then solves the resulting set covering problem to construct a good feasible solution for the original problem. We conduct computational experiments for instances with various constraints and confirm the flexibility and robustness of our algorithm.

1 Introduction

The pickup and delivery problem with time windows (PDPTW) is a problem that asks to find optimal routes and schedules of a fleet of vehicles serving all requests [5, 15]. Each request signifies the delivery of a demand from an origin to a destination. The origin and destination of each request must be visited by the same vehicle in the order of origin and destination. Each service (i.e., pickup at an origin or delivery at a destination) must start within a given time window (time window constraint). Each vehicle has a capacity, and the total amount of loads of a vehicle must always be kept within its capacity (capacity constraint).

Exact and heuristic algorithms for this problem has widely being studied. Savelsbergh and Sol [14] proposed a branch and price algorithm based on a set partitioning formulation. Dumas, Desrosiers and Soumis [6] proposed a column generation scheme using a constrained shortest path as a subproblem. Nanry and Barnes [10] presented a reactive tabu search approach. A variant of the genetic algorithm called a grouping genetic algorithm was presented by Pankratz [11]. Li and Lim [9] proposed a tabu-embedded simulated annealing. They also generated new benchmark instances, and tested the performance of their algorithm on them. Bent and van Hentenryck [2] and Ropke and Pisinger [13] proposed large neighborhood search based algorithms, and obtained good results on the benchmarks of Li and Lim.

In this paper, we further generalize the pickup and delivery problem with time windows by allowing additional constraints on each route (abbreviated as PDP-ACER) such as the Last-in-First-Out constraint (abbreviated as LIFO), renewable or nonrenewable multi resources and so on. The LIFO constraint says that a load being picked up is always placed at the rear of the vehicle while only the load at the rear can be unloaded. As these constraints are diverse, it is not realistic to develop solution methods in individual cases. Hence we try to develop a method which treats those constraints in an integrated way, where we assume that all constraints have the monotone property:

If a route covering a set of requests satisfies a given constraint, then any subroute (i.e., covering a subset of the requests) also satisfies the constraint.

We note that many constraints that appear in practice are monotone, and that their feasibility can be determined easily. The LIFO is an example of such constraints. Cordeau et al. [4] and Carrabs et al. [3] addressed the pickup and delivery traveling salesman problem with the LIFO constraints. If we assume that the traveling times satisfy the triangle inequalities, then it implies that time window constraints also satisfy the monotone property.

In our approach, we formulate the problem as a set covering problem (abbreviated as SCP), such that all requests must be covered by a set of feasible routes. Since enumerating all feasible routes is not realistic, we try to construct a set of good feasible routes which is of manageable size, but has sufficient diversity. It constructs an initial set of routes by the insertion method, and then repeats reconstructing the set of candidate routes. In the reconstruction procedure, we estimate the attractiveness of a route by its relative cost of the Lagrangian relaxation of the set covering problem with the current set of routes. It then generates new routes from those with small relative costs by applying five types of operations. The resulting SCP instance is then solved to find a good feasible solution of PDP-ACER. Although a solution of SCP may cover a request more than once, we can easily transform it into a feasible solution of the original problem as a result of the monotone property of constraints. This type of approach, called column generation, is known to be useful for problems with complicated or tight constraints. Note that our algorithm is heuristic though the column generation method is usually used for exact algorithms. For PDPTW, Savelsbergh and Sol [14] and Dumas, Desrosiers and Soumis [6] proposed exact algorithms using the column generation approach based on a set partitioning formulation of the problem.

To confirm the flexibility and efficiency of our algorithm, we conducted computational experiments. We first confirmed the usefulness of using the Lagrangian relaxation, and then tested our algorithm on available benchmark instances of PDPTW as well as some new instances with additional and/or modified constraints. We compared our algorithm with a local search type algorithm which we prepared for the purpose of comparison, and confirmed the flexibility of our algorithm.

2 Problem Definition

We formulate PDP-ACER as follows. Let $G = (V, E)$ be a complete directed graph with vertex set $V = \{0, 1, \ldots, 2n\}$ and edge set $E = \{(i, j) \mid i, j \in V, i \neq j\}$. In this graph, vertex 0 is the depot. Vertices from 1 to n are customers where loads are picked up and vertices from $n+1$ to $2n$ are customers where loads are delivered. Each edge $(i, j) \in E$ has a traveling cost $c_{ij} \geq 0$ and a traveling time $t_{ij} \geq 0$. The traveling costs and times satisfy the triangle inequalities,

$$c_{ik} + c_{kj} \geq c_{ij} \text{ and } t_{ik} + t_{kj} \geq t_{ij}, \ \forall i, j, k \in V. \tag{1}$$

Let $H = \{1, 2, \ldots, n\}$ be a given set of requests. Each request $h \in H$ signifies the delivery from the origin $h \in V$ to the destination $h + n \in V$ (for convenience, we call a request and its origin by the same name h). The vertices h and $h + n$ must be visited by the same vehicle (coupling constraint), and h must be visited before $h + n$ (precedence constraint). All requests are served by a fleet of homogeneous vehicles. Each vehicle must start from the depot, serve some requests and return to the depot. Let S_r be the set of requests served by its route r, $m_r = |S_r|$, and σ_r be the sequence of customers to be visited, where $\sigma_r(k)$ denotes the kth customer in the route r. We assume $\sigma_r(0) = \sigma_r(2m_r + 1) = 0$ for convenience.

In this paper, we consider various constraints imposed on each route. Each customer $i \in V$ has a handling time s_i for the service and a time window $[e_i, l_i]$, where e_i is the release time to serve i and l_i is the deadline of the service. Serving a request h consumes the resource, which are classified into renewable and nonrenewable resources. For example,

the weight of loads can be treated as renewable resources, and the workload for pickup and delivery of loads can be treated as nonrenewable resources. Each request h consumes q_{hp}^{re} units of renewable resources ($p = 1, 2, \ldots, \rho$) while it is loaded, and consumes $q_{hp'}^{\mathrm{non}}$ units of nonrenewable resources ($p' = 1, 2, \ldots, \pi$). Each vehicle has capacities Q_p^{re} for renewable resources p and $Q_{p'}^{\mathrm{non}}$ for nonrenewable resources p'. The total load of each renewable resource p at each customer k in route r must not exceed the capacity Q_p^{re}; i.e.,

$$\sum_{h \in S_r : \sigma_r^{-1}(h) \le k < \sigma_r^{-1}(h+n)} q_{hp}^{\mathrm{re}} \le Q_p^{\mathrm{re}} \text{ for any } k = 0, 1, \ldots, 2m_r.$$

The total load of each nonrenewable resource p' in route r must be within $Q_{p'}^{\mathrm{non}}$; i.e.,

$$\sum_{h \in S_r} q_{hp'}^{\mathrm{non}} \le Q_{p'}^{\mathrm{non}}.$$

We further introduce the LIFO constraint. That is, if a request h is picked up before a request h', either h is delivered before the pickup of h' or after the delivery of h'; i.e., $\sigma_r^{-1}(h) < \sigma_r^{-1}(h')$ implies either

$$\sigma_r^{-1}(h) < \sigma_r^{-1}(h') < \sigma_r^{-1}(h'+n) < \sigma_r^{-1}(h+n)$$

or

$$\sigma_r^{-1}(h) < \sigma_r^{-1}(h+n) < \sigma_r^{-1}(h') < \sigma_r^{-1}(h'+n).$$

The standard PDPTW has the time window constraint and only the one dimensional renewable resource (i.e., $\rho = 1$ and $\pi = 0$). In this paper, we permit the time window constraint and and more general resource constraints (i.e., $\rho > 1$ and $\pi > 0$ are allowed). As for the LIFO constraint, we consider both cases in which it is imposed and not. In addition to the above constraints, any monotone constraint can be imposed, assuming that we have an algorithm to efficiently test its feasibility. We remark that the following property holds under monotone constraints.

Property 2.1 *Given a feasible route, any request can be deleted from the route without violating the constraints and without increasing the cost.*

Let ν be the number of vehicles used in a solution. A feasible solution is a set $\{\sigma_1, \sigma_2, \ldots, \sigma_\nu\}$ of routes such that each σ_r satisfies all the given constraints and each request is serviced exactly once. In the literature, it is often considered that the primary objective is to reduce the number of vehicles, and the secondary objective is to minimize the total traveling cost. However, for convenience, we adopt the following objective function:

$$\sum_{r=1}^{\nu} C_r,$$

where

$$C_r = \alpha + \sum_{i=0}^{2m_r} c_{\sigma_r(i)\sigma_r(i+1)}$$

(i.e., C_r is the sum of the fixed cost α for using one vehicle and the traveling cost of r). If we need to reduce the number of vehicles, we set α to a large value compared with the traveling cost.

3 Set Covering Formulation

The PDP-ACER can be formulated as the following set covering problem:

$$\begin{array}{lll} \text{SCP}(R^*) & \text{minimize} & \sum_{r \in R^*} C_r x_r \\ & \text{subject to} & \sum_{r \in R^*} a_{hr} x_r \geq 1, \quad \forall h \in H \\ & & x_r \in \{0,1\}, \quad \forall r \in R^* \end{array}$$

where R^* is the set of all feasible routes, and

$$a_{hr} = \begin{cases} 1 & \text{(if request } h \text{ is in route } r \in R^*) \\ 0 & \text{(otherwise).} \end{cases}$$

Note that in this formulation we can write $\sum_{r \in R^*} a_{hr} x_r \geq 1$ instead of $\sum_{r \in R^*} a_{hr} x_r = 1$ by Property 2.1.

However, enumerating all feasible routes is not realistic because the size of R^* is exponentially large. We therefore choose a subset R ($\subseteq R^*$) of manageable size and solve the corresponding set covering problem SCP(R). The obtained solution may not be an optimal solution to SCP(R^*) but is a feasible solution. If R is cleverly constructed to represent R^*, the solution would be a good feasible solution to SCP(R^*). In order to solve SCP(R), we use the algorithm proposed by Yagiura et al. [16]. Finally we construct a solution of PDP-ACER from the solution of SCP(R). The solution to SCP(R) may contain more than one route serving the same request. In this case, based on Property 2.1, we can remove the over-covered requests one by one in a greedy way until no such request remains.

The following is the outline of our algorithm:

1. Generate a set R of feasible routes.
2. Solve the resulting instance of SCP(R).
3. Construct a feasible solution of PDP-ACER from the solution obtained in 2.

The main part of our algorithm is how to generate the set R. To obtain a good solution, we need to choose R very carefully. For instance, if we generate a large set R that has only similar routes, it will take a large amount of time to solve SCP(R) and the quality of a solution may be poor. On the other hand, if we can construct a small set R of good routes having sufficient diversity, then we can expect to get a good solution in short computation time. The route generation will be described in Section 4.

4 Route Generation

Our route generation algorithm consists of two phases. The first phase is the initial construction phase, which generates a certain number of routes for each request by an insertion method. The second phase is the reconstruction phase, which chooses good routes from the current set of routes, and add their neighboring routes. To estimate the attractiveness of a route, we use its relative cost of the Lagrangian relaxation of SCP(R), where R is the current set of routes. The algorithm executes the initial construction phase once, and then repeats the reconstruction phase until a given time limit is reached.

The algorithm may possibly generate duplicate routes in the sense of covering the same set of requests. To avoid such duplication, we use a hash table, and check whether such a route already exists in R or not, whenever a new route is added in R. If a route with the same set of requests exits, we choose the one having the lower cost.

4.1 Initial Construction Phase

The initial construction phase starts from the empty set $R = \emptyset$, and applies an insertion method to generate β (a parameter) routes for each request. The insertion method first prepares a route that contains only the specified request and the depot, and then repeats inserting requests into the route by the criteria as described below, as far as the feasibility of constraints is maintained. When the route becomes maximal (i.e., no more request can be inserted to it), we add it to R.

The insertion method proceeds as follows. We define the insertion cost of a request h into route r, when its origin h is inserted between $\sigma_r(k)$ and $\sigma_r(k+1)$ and its destination $n+h$ is inserted between $\sigma_r(k')$ and $\sigma_r(k'+1)$ $(k' \geq k)$, by

$$\delta_r(h,k,k') = \begin{cases} c_{\sigma_r(k)h} + c_{h,h+n} + c_{h+n,\sigma_r(k+1)} - c_{\sigma_r(k)\sigma_r(k+1)}, & \text{if } k = k' \\ c_{\sigma_r(k)h} + c_{h\sigma_r(k+1)} - c_{\sigma_r(k)\sigma_r(k+1)} & \\ \quad + c_{\sigma_r(k'),h+n} + c_{h+n,\sigma_r(k'+1)} - c_{\sigma_r(k')\sigma_r(k'+1)}, & \text{otherwise.} \end{cases}$$

We then define $\delta_r^{\min}(h)$ as the minimum of $\delta_r(h,k,k')$ among all k and k' whose resulting routes are feasible. If all combinations of k and k' are infeasible, we set $\delta_r^{\min}(h) = \infty$. If request h is chosen and $\delta_r^{\min}(h) < \infty$, we thus insert h to the best positions k and k' which attains $\delta_r^{\min}(h) = \delta_r(h,k,k')$. Next we describe how to choose requests h to insert. If the algorithm always chooses the request that achieves the minimum insertion cost, the resulting set of routes may not have sufficient diversity, which is not desirable in order to achieve high performance. We therefore incorporate randomness in the manner as often used in GRASP (greedy randomized adaptive search procedure) [7]. Let D_r be the set of requests h with the κ (κ is a parameter) smallest values of $\delta_r^{\min}(h)(< \infty)$ among those in $H \setminus S_r$ (i.e., the requests not in route r). Then, in each iteration, the algorithm chooses a request h randomly from D_r, until a maximal route is reached. In this way, we usually obtain different routes by this insertion method, even if it starts from the same initial request. Let Construct(β) be the set of routes output in this phase.

4.2 Reconstruction Phase

In the reconstruction phase, it modifies the given set of routes by using the Lagrangian relaxation of the set covering problem SCP(R). It first calculates the Lagrangian multipliers by applying a subgradient method, and, based on them, selects some routes from the current set R (Section 4.2.1). Then it generates additional routes by applying five types of operations to the selected routes, and updates the set R (Section 4.2.2 and 4.2.3). This procedure is repeated until no new route is generated or until it reaches the time limit.

4.2.1 Selection of Routes

From the current set R, the algorithm selects some number of routes for two purposes: (1) to choose a set of routes from which new routes are generated, and (2) to reduce the number of routes in R when the size of R becomes too large. We estimate the attractiveness of a route by its relative cost for the Lagrangian relaxation problem of SCP(R). See for example the review by Fisher [8] for the Lagrangian relaxation.

The Lagrangian relaxation of SCP(R) with a given nonnegative $n = |H|$ dimensional Lagrangian multiplier vector $\boldsymbol{u} = (u_1, u_2, \ldots, u_n)$ is defined as follows:

$$\begin{aligned} L(\boldsymbol{u}) &= \min_{\boldsymbol{x}\in\{0,1\}^{|R|}} \sum_{r\in R} C_r x_r + \sum_{h\in H} u_h \left(1 - \textstyle\sum_{r\in R} a_{hr} x_r\right) \\ &= \min_{\boldsymbol{x}\in\{0,1\}^{|R|}} \sum_{r\in R} c_r(\boldsymbol{u}) x_r + \sum_{h\in H} u_h, \end{aligned} \tag{2}$$

where

$$c_r(\boldsymbol{u}) = C_r - \sum_{h\in H} a_{hr} u_h$$

is the relative cost associated with r. An optimal solution $\boldsymbol{x}(\boldsymbol{u})$ to problem (2) is easily obtained by

$$x_r(\boldsymbol{u}) = \begin{cases} 1 & \text{if } c_r(\boldsymbol{u}) < 0 \\ 0 \text{ or } 1 & \text{if } c_r(\boldsymbol{u}) = 0 \\ 0 & \text{if } c_r(\boldsymbol{u}) > 0. \end{cases}$$

The value $L(\boldsymbol{u})$ gives a lower bound on the optimal value of problem SCP(R). The Lagrangian dual is the problem of finding a Lagrangian multiplier vector $\boldsymbol{u}^*$ that maximizes $L(\boldsymbol{u})$. It is known that an optimal multiplier vector $\boldsymbol{u}^*$ can be obtained as an optimal solution to the dual of the LP relaxation of SCP:

$$\begin{aligned} \text{maximize} \quad & \sum_{h \in H} u_h \\ \text{subject to} \quad & \sum_{h \in H} u_h a_{hr} \leq C_r, \quad \forall r \in R \\ & u_h \geq 0, \qquad\qquad \forall h \in H. \end{aligned}$$

If a good Lagrangian multiplier vector $\boldsymbol{u}$ is obtained, the relative cost $c_r(\boldsymbol{u})$ gives reliable information on the attractiveness of fixing $x_r = 1$, because it is reported that all r with $x_r = 1$ in an optimal solution of SCP tend to have small $c_r(\boldsymbol{u})$ values.

We calculate the Lagrangian multiplier $\boldsymbol{u}$ for SCP(R) by a heuristic approach called the subgradient method [1, 8, 16], because computing an optimal $\boldsymbol{u}^*$ of the above LP problem is usually quite expensive. We evaluate a route r by its relative cost $c_r(\boldsymbol{u})$ of the obtained Lagrangian multiplier $\boldsymbol{u}$. Let R' be the set of routes with an (a is a parameter) smallest values of $c_r(\boldsymbol{u})$ among those in R. Furthermore, for each request $h \in H$, let R''_h be the set of routes with the b (b is a parameter) smallest values of $c_r(\boldsymbol{u})$ among those in R that include h. Finally let $R'' = \bigcup_{h \in H} R''_h$. Our procedure Selection(R, $\boldsymbol{u}$, a, b) outputs the set $R' \cup R''$.

4.2.2 Neighboring Routes of a Route

We introduce three operations to generate neighboring routes of a route r.

Insertion This operation inserts a new request h into r at the best position (i.e., at the pair of positions that achieves $\delta_r^{\min}(h)$). The algorithm applies this operation for each request (which is not in r), and all feasible routes obtained by these operations are output. Let Insertion(r) be the set of routes output by applying this procedure to r, whose size is $|\text{Insertion}(r)| = O(n)$.

Deletion This operation deletes one request from r. The algorithm applies this operation for each request in r, and all routes obtained by these operations are output. Note that the feasibility after deletion is preserved by Property 2.1. Let Deletion(r) be the set of routes output by applying this procedure to r, whose size is $|\text{Deletion}(r)| = O(m_r)$.

Swap This operation deletes one request from r and then inserts one request which is not in r at the best position. The algorithm applies this operation for all pairs of a request in r and another not in r. All feasible routes obtained by these operations are output. Let Swap(r) be the set of routes output by applying this procedure to r, whose size is $|\text{Swap}(r)| = O(m_r n)$.

4.2.3 Neighboring Routes of Two Routes

In addition, we use two operations to generate neighboring routes of two routes r and r'.

2-opt* method This operation is similar to the 2-opt* neighborhood operation proposed by Potvin et al. [12]. Given two routes r and r' satisfying $S_r \cap S_{r'} = \emptyset$, it first constructs a route by concatenating the former part of r and the latter part of r', cut at k and k':

$$(\sigma_r(0), \sigma_r(1), \ldots, \sigma_r(k), \sigma_{r'}(k'), \sigma_{r'}(k'+1), \ldots, \sigma_{r'}(2m_{r'}+1)).$$

For this, it chooses a random position k of r, and then chooses the minimum k' such that the resulting concatenated route is feasible with respect to the time window constraint. However, the resulting route may not satisfy the coupling or other constraints, and some modification may be necessary for remedy. To recover the coupling constraints, for example, it inserts for each violating customer in the route the corresponding customer not in the route at the best position under the feasibilities of other constraints. Otherwise it deletes the violating customer from the route. Similar remedies are applied to recover other constraints. It repeats this process until all requests in the route satisfy the given constraints. Let 2-opt*(r, r') be the generated route by applying this procedure to r and r' if it exists; otherwise it denotes the empty set.

Mixing two routes Given two routes r and r', and a Lagrangian multiplier vector $\boldsymbol{u}$, this operation starts from $\sigma_{\mathrm{mix}} := \sigma_r$ and repeats modifying the current σ_{mix} so that its set of requests becomes closer to that of $\sigma_{r'}$, by inserting or deleting different requests between σ_{mix} and $\sigma_{r'}$. Similarly to $\delta_r^{\min}(h)$, we denote by $\delta_{\mathrm{mix}}^{\min}(h)$ the minimum increase in the cost when request h is inserted into σ_{mix}. In each iteration, an insertion is first tried: It chooses the request h that minimizes $\delta_{\mathrm{mix}}^{\min}(h) - u_h$ (i.e., the increase in the relative cost) among those requests which are in $\sigma_{r'}$ but not in σ_{mix}, and inserts it at the best position of σ_{mix} provided that the resulting route is feasible. If there is no such request or all inserting positions make the resulting route infeasible for all such requests, then it turns to the deletion operation with the following rule. Let

$$\delta_{\mathrm{mix}}^{-}(h) = \begin{cases} c_{\sigma_{\mathrm{mix}}(k-1)\sigma_{\mathrm{mix}}(k+2)} - c_{\sigma_{\mathrm{mix}}(k-1),h} & \\ \quad - c_{h,h+n} - c_{h+n,\sigma_{\mathrm{mix}}(k+2)}, & \text{if } k' = k+1 \\ c_{\sigma_{\mathrm{mix}}(k-1)\sigma_{\mathrm{mix}}(k+1)} + c_{\sigma_{\mathrm{mix}}(k'-1)\sigma_{\mathrm{mix}}(k'+1)} & \\ \quad - c_{\sigma_{\mathrm{mix}}(k-1),h} - c_{h,\sigma_{\mathrm{mix}}(k+1)} & \\ \quad - c_{\sigma_{\mathrm{mix}}(k'-1),h+n} - c_{h+n,\sigma_{\mathrm{mix}}(k'+1)}, & \text{otherwise} \end{cases}$$

where $\sigma_{\mathrm{mix}}(k) = h$ and $\sigma_{\mathrm{mix}}(k') = h + n$. Then the operation chooses the request h with the minimum $\delta_{\mathrm{mix}}^{-}(h) + u_h$ (i.e., the increase of the relative cost) among those not in $\sigma_{r'}$ but in σ_{mix}, and removes it from σ_{mix}.

Letting σ_{mix} be the new route obtained either by the insertion or the deletion, the algorithm executes another iteration (that starts with insertion and then deletion if insertion is impossible) unless $\sigma_{\mathrm{mix}} = \sigma_{r'}$ holds.

All routes obtained during the above modifications are considered as candidates to be added into R. Let Mixing$(r, r', \boldsymbol{u})$ be the set of all feasible routes output by this procedure from routes r and r'. Its size is $\left|\mathrm{Mixing}(r, r', \boldsymbol{u})\right| = O(m_r + m_{r'})$.

4.2.4 Reconstruction Algorithm

The entire reconstruction algorithm by the above five operations is summarized as follows.

Algorithm Reconstruction(R, a, b, a', b', μ)

Input: A set R of routes, parameters a, b, a', b' and μ.

Output: A set R' of routes.

Step 1. Calculate the Lagrangian multiplier $\boldsymbol{u}$ by the subgradient method.

Step 2. Let $\hat{R}$:=Selection$(R, \boldsymbol{u}, a, b)$ and $R' := R$.

Step 3. Let $R' := R' \cup \left(\cup_{r \in \hat{R}}(\mathrm{Insertion}(r) \cup \mathrm{Deletion}(r) \cup \mathrm{Swap}(r))\right)$

Step 4. For all pairs of routes $r, r' \in \hat{R}$,

$$R' := \begin{cases} R' \cup \text{2-opt}^*(r, r') & \text{if } S_r \cap S_{r'} = \emptyset \\ R' \cup \mathrm{Mixing}(r, r', \boldsymbol{u}) & \text{otherwise.} \end{cases}$$

Step 5. If $|R'| > \mu$, let $R' :=$ Selection(R', $\boldsymbol{u}$, a', b').

Step 6. Return R'.

As described before, the algorithm reconstructs the set of routes by calling algorithm Reconstruction repeatedly until it reaches a given time limit.

4.3 Overall Algorithm

Let ζ be an upper limit of computation time of constructing routes. We use a heuristic SCP solver by Yagiura et al. [16] (denoted YKI) whose time limit can be set arbitrarily. Let ζ' be an upper limit of computation time of YKI. Then overall algorithm is described as follows:

Algorithm RouteGeneration($\mathcal{I}, \zeta, \zeta', \beta, \mu, a, b, a', b'$)

Input: A PDP-ACER instance $\mathcal{I}$, parameters $\zeta, \zeta', \beta, \mu, a, b, a'$ and b'.

Output: A set R of routes.

Step 1. Let R' :=Construction(β) and $rep := 0$.

Step 2. If total computing time reaches ζ, then go to Step 4.

Step 3. R' :=Reconstruction(R', a, b, a', b', μ) and $rep := rep + 1$. Return to Step 2.

Step 4. Convert R' into an instance of SCP, and solve it by YKI with time limit ζ'. Let $\hat{R}$ be the output solution of the SCP.

Step 5. Construct a solution R of the PDP-ACER from $\hat{R}$.

Step 6. Return R.

5 Computational Experiment

We conducted computational experiments to evaluate the proposed algorithm, which was coded in C and run on a PC (Intel Pentium4, 2.8 GHz, 1 GB memory). We used the instance groups having 100 to 400 customers from the PDPTW benchmarks of Li and Lim [9]. The instances are categorized into the type-C1, C2, R1, R2, RC1, RC2. The types C, R and RC represent the distribution of the customers in each instance. The customers are distributed as clusters in type-C and distributed randomly in type-R. In type-RC, the customers are partially distributed as clusters and the rest is distributed randomly. The types 1 and 2 represent the severeness of the time window and the capacity constraints of the instances; the type 1 instances have severer constraints than the type 2 instances (hence more vehicles are needed). The instances with 100 customers consist of 9 type-C1 instances, 12 type-R1 instances, 8 type-RC1 instances, 8 type-C2 instances, 11 type-R2 instances and 8 type-RC2 instances. The instances with 200 and 400 customers consist of 10 instances for each of type-C1, C2, R1, R2, RC1, RC2.

5.1 Efficiency of Using Lagrangian Multiplier

In the reconstruction phase of the route generation, relative cost is used to choose a subset $\hat{R}$ ($\subseteq R$) for generating new routes. To confirm the effectiveness of this approach, we tested two other methods for selecting a set of routes in the reconstruction phase. For comparison purpose, we solved SCP(R) with the algorithm YKI whenever algorithm Reconstruction outputs R, and observe the quality of the solution. The first method selects the set of routes appearing in the best solution of SCP(R) found by YKI, and the second method selects a set of routes randomly from the current R. We conducted the comparison of these two methods with the method in Section 4.2 that uses the relative cost.

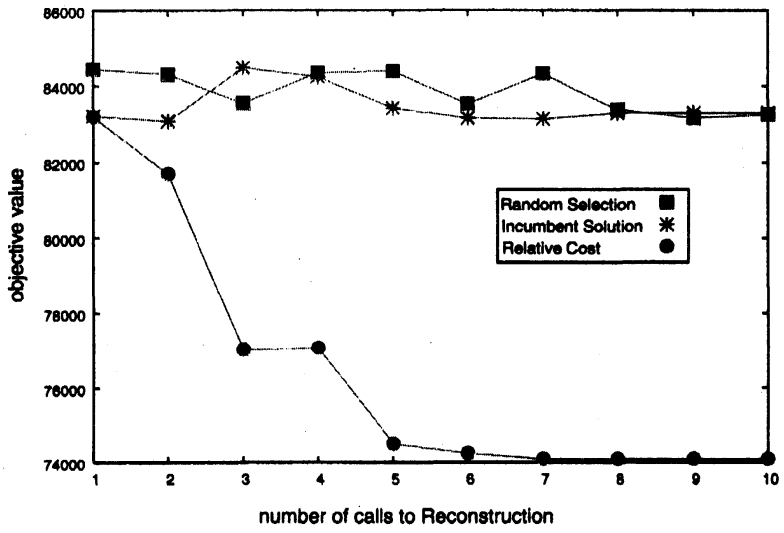

Figure 1: Comparison of the three selection methods of routes (type-C instance)

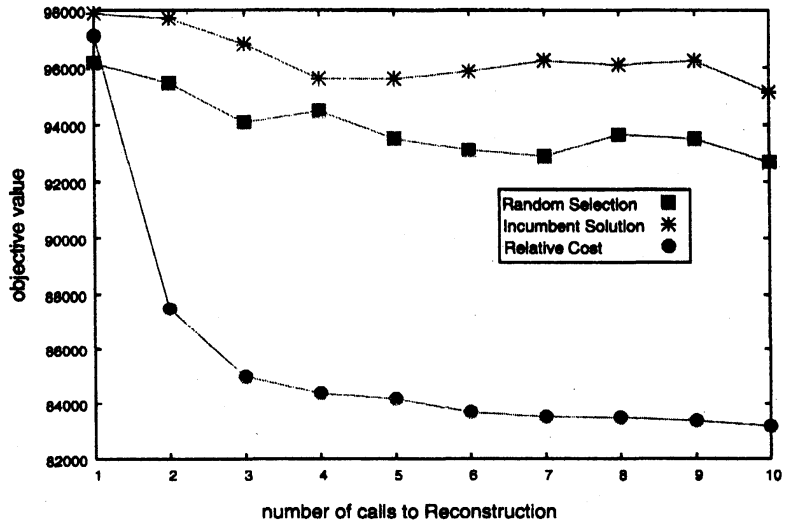

Figure 2: Comparison of the three selection methods of routes (type-R instance)

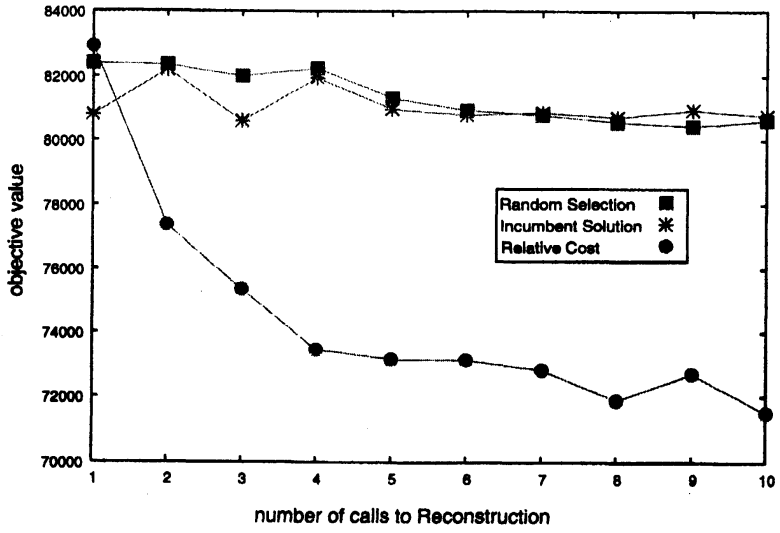

Figure 3: Comparison of the three selection methods of routes (type-RC instance)

Figures 1, 2 and 3 show the objective values of the solutions of SCP(R) obtained by YKI against the number of iterations of algorithm Reconstruction. Figure 1 shows the result on a type-C instance, and Figure 2 shows the result on a type-R instance, Figure 3 is the result on a type-RC instance. In all figures, as the number of calls to Reconstruction increases, the results of "Relative Cost" become better than the others. We therefore adopted the method based on the relative cost in the algorithm Reconstruction.

5.2 Results on Benchmark Instances

Next, we tested our algorithm on the PDPTW benchmarks of Li and Lim [9] as explained in the beginning of Section 5. We set parameters to $\beta = 200$, $a = 3$, $b = 4$, $a' = 150,000$, $b' = 600,000/|H|$ and $\mu = 600,000$. The time limit ζ of constructing routes (i.e., excluding the time for solving the set covering problem) is set to 600 seconds for the instances with 100 customers, 1400 seconds for the instances with 200 customers and 2500 seconds for the instances with 400 customers. We set the time limit ζ' for solving a set covering instance to 400 seconds for 100 customers, 600 seconds for 200 customers and 1500 seconds for 400 customers instances. Therefore, in total, we spend 1000, 2000 and 4000 seconds for the instances with 100, 200 and 400 customers, respectively. Table 1 shows the results of our algorithm in column "Ours", and the one proposed by Ropke and Pisinger [13] in "RP". Their algorithm is based on Large Neighborhood Search. They ran their algorithm for each instance ten times on a 1.5 GHz PC with 256 MB memory. We compare our results with their average results of the ten runs. In Table 1, column "$2n$" represents the number of customers in the instance group and column "type" represents the type of the instance group. Columns "CNV" and "CDIST" represent the cumulative number of vehicles and the cumulative traveling cost for the instances. Column "TIME" of "Ours" represents the computation time in seconds for each instance and that of "RP" represents the average computation time.

In Table 1, we observe that our method could not obtain better results than those of Ropke and Pisinger. Note that the algorithm of Ropke and Pisinger is specialized to the PDPTW while our algorithm can treat a variety of constraints. For type 2 instances, the difference in solution quality is large, while for type 1 instances (having severer constraints than type 2), the difference is rather small both in the number of vehicles and in the traveling cost.

Table 1: Results on Li and Lim's instances

		Ours			RP		
$2n$	type	CNV	CDIST	TIME	CNV	CDIST	TIME
100	1	322	33650.65	1000	322.0	33599.02	41
100	2	85	29557.69	1000	81.0	24650.45	92
200	1	470	103763.05	2000	469.1	100940.60	158
200	2	150	100435.79	2000	139.0	80766.76	369
400	1	914	258333.55	4000	904.4	241015.00	543
400	2	311	250065.39	4000	263.4	184801.80	1219

Table 2: Constraints of instances

	Resource		Capacity			
INSTANCE	ρ	π	$Q1$	$Q2$	TW	LIFO
GC1	1	0	200	1000	$[e_i, l_i]$	0
GC2	3	1	200	1000	$[e_i, l_i]$	1
GC3	1	0	200	1000	$[e'_i, l'_i]$	0
GC4	1	1	200	1000	$[0, \infty)$	0
GC5	1	1	200	1000	$[0, \infty)$	1
GC6	2	0	200	200	$[e_i, l_i]$	0

Table 3: Comparison for GC1–GC6

	Ours		LS	
INSTANCE	CNV	CDIST	CNV	CDIST
GC1	208	65624.54	224	72422.65
GC2	278	95016.41	313	92170.04
GC3	142	48421.68	155	56234.36
GC4	234	79763.98	212	59545.98
GC5	238	84378.57	212	55065.95
GC6	271	84785.49	276	82716.75

5.3 Comparison of Our Algorithm with a Metaheuristic Algorithm

Finally, we conducted experiments to confirm the flexibility and performance of our algorithm. We compared our algorithm with a metaheuristic algorithm coded in reference to the algorithm proposed for PDPTW by Li and Lim [9]. It is based on a simulated annealing and tabu search procedure, which uses the same objective function as ours; that is, the primary objective is to reduce the number of vehicles and the secondary objective is to minimize the total traveling cost. We modify it so that it can deal with PDP-ACER. The modified algorithm executes the local search in a feasible region of the constraints of PDP-ACER.

We generated the PDP-ACER instances consisting of six groups GC1–GC6, modified from the PDPTW instances of Li and Lim [9] by adding various constraints. We chose three instances from those of Li and Lim for each type, and generated new instances from them; hence each of GC1–GC6 contains 18 instances. Table 2 gives a sketch of the constraints of those groups. In Table 2, columns "ρ" and "π" represent the number of renewable and nonrenewable resources. Column "$Q1$" (resp., "$Q2$") represents the vehicle capacities of type 1 (resp., type 2) instances; that is, we set $Q_p^{\mathrm{re}} := Q1$ and $Q_{p'}^{\mathrm{non}} := Q1$ (resp., $Q_p^{\mathrm{re}} := Q2$, $Q_{p'}^{\mathrm{non}} := Q2$) for all type 1 (resp., type 2) instances. Column "TW" shows the information about the time window constraint. In GC4 and 5, we set all time windows to $[0, \infty)$ (i.e., no time window constraints). On the other hand, in GC3, we cut 4% from the original time windows by setting $[e_i, l_i]$ to $[e_i', l_i']$ such that

$$\begin{aligned} e_i' &= e_i + 0.02(l_i - e_i), \\ l_i' &= l_i - 0.02(l_i - e_i), \quad \forall i \in V. \end{aligned}$$

For the rest (i.e., GC1, GC2 and GC6), we adopted the time windows of the original instances. We imposed the LIFO constraint to GC2 and GC5 as shown in the LIFO column by 1.

We set parameters to $\beta = 200$, $a = 3$, $b = 4$, $a' = 150,000$, $b' = 600,000/|H|$ and $\mu = 600,000$. The time limit of constructing routes is set to 2400 seconds and the time limit of solving the set covering problem is set to 1200 seconds. We set the time limit to 3600 seconds for the metaheuristic algorithm. Table 3 compares the results of our algorithm and those of metaheuristic algorithm. In Table 3, column "INSTANCE" represents the name of each instance group, column "CNV" means the cumulative number of vehicles and column "CDIST" means the cumulative traveling cost.

The results show that for GC1, GC2, GC3 and GC6 whose instances have additional constraints or tight constraints, our algorithm works efficiently, but for GC4 and GC5 whose instances have weaker constraints, the metaheuristic algorithm works better than ours. These results confirm our expectation that our algorithm works well on the instances with tighter constraints, because the number of feasible routes is limited in such cases.

6 Conclusion

We generalized the pickup and delivery problem with time windows by allowing additional constraints having monotone property. Our algorithm first generates a set of feasible routes and then solves the resulting set covering problem. We construct an initial set of routes by an insertion method, and reconstruct the resulting set repeatedly by using various types of neighborhood operations, while reducing the set size of candidate routes by utilizing the Lagrangian relative costs. The computational results indicated that our algorithm works more efficiently than a metaheuristic algorithm, if the instances have tighter constraints. We also confirmed the flexibility of our algorithm by applying it to instances with various constraints.

References

[1] E. Balas and A. Ho. Set covering algorithms using cutting planes, heuristics, and subgradient optimization: a computational study. *Mathematical Programming Study*, 12:37–60, 1980.

[2] R. Bent and P. Van Hentenryck. A two-stage hybrid algorithm for pickup and delivery vehicle routing problems with time windows. *Computers and Operations Research*, 33:875–893, 2006.

[3] F. Carrabs, J.-F. Cordeau, and G. Laporte. Variable neighborhood search for the pickup and delivery traveling salesman problem with LIFO loading. *INFORMS Journal on Computing*, to appear.

[4] J.-F. Cordeau, M. Iori, G. Laporte, and J. J. S. González. A branch-and-cut algorithm for the pickup and delivery traveling salesman problem with LIFO loading. *Networks*, to appear.

[5] J. Desrosiers, Y. Dumas, M. M. Solomon, and F. Soumis. Time constrained routing and scheduling. In M. O. Ball, T. L. Magnanti, C. L. Monma, and G. L. Nemhauser, editors, *Network Routing*, volume 8 of *Handbooks in Operations Research and Management Science*, pages 35–139. North-Holland, Amsterdam, 1995.

[6] Y. Dumas, J. Desrosiers, and F. Soumis. The pickup and delivery problem with time windows. *European Journal of Operational Research*, 54:7–22, 1991.

[7] T. A. Feo and M. G. C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[8] M. L. Fisher. The Lagrangian relaxation method for solving integer programming problems. *Management Science*, 27(1):1–18, 1981.

[9] H. Li and A. Lim. A metaheuristic for the pickup and delivery problem with time windows. *International Journal on Artificial Intelligence Tools*, 12(2):173–186, 2003.

[10] W. P. Nanry and J. W. Barnes. Solving the pickup and delivery problem with time windows using reactive tabu search. *Transportation Research Part B*, 34:107–121, 2000.

[11] G. Pankratz. A grouping genetic algorithm for the pickup and delivery problem with time windows. *OR Spectrum*, 27:21–41, 2005.

[12] J.-Y. Potvin, T. Kervahut, B.-L. Garcia, and J.-M. Rousseau. The vehicle routing problem with time windows part I: tabu search. *INFORMS Journal on Computing*, 8(2):158–164, 1996.

[13] S. Ropke and D. Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. Technical report, Department of Computer Science, University of Copenhagen, 2004.

[14] M. Savelsbergh and M. Sol. Drive: Dynamic routing of independent vehicles. *Operations Research*, 46(4):474–490, 1998.

[15] P. Toth and D. Vigo, editors. *The Vehicle Routing Problem.* Society for Industrial and Applied Mathematics, 2002.

[16] M. Yagiura, M. Kishida, and T. Ibaraki. A 3-flip neighborhood local search for the set covering problem. *European Journal of Operational Research*, 172:472–499, 2006.